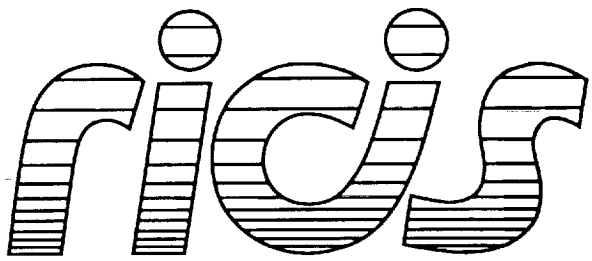
Advanced Software Development Workstation Project Engineering Scripting Language Graphical Editor SUMMARY REPORT

Inference Corporation

2/14/92

Cooperative Agreement NCC 9-16
Research Activity No. SE.41

NASA Johnson Space Center
Information Systems Directorate
Information Technology Division



Research Institute for Computing and Information Systems
University of Houston-Clear Lake

TECHNICAL REPORT

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information Systems (RICIS) in 1986 to encourage the NASA Johnson Space Center (JSC) and local industry to actively support research in the computing and information sciences. As part of this endeavor, UHCL proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a continuing cooperative agreement with UHCL beginning in May 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The UHCL/RICIS mission is to conduct, coordinate, and disseminate research and professional level education in computing and information systems to serve the needs of the government, industry, community and academia. RICIS combines resources of UHCL and its gateway affiliates to research and develop materials, prototypes and publications on topics of mutual interest to its sponsors and researchers. Within UHCL, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business and Public Administration, Education, Human Sciences and Humanities, and Natural and Applied Sciences. RICIS also collaborates with industry in a companion program. This program is focused on serving the research and advanced development needs of industry.

Moreover, UHCL established relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research. For example, UHCL has entered into a special partnership with Texas A&M University to help oversee RICIS research and education programs, while other research organizations are involved via the "gateway" concept.

A major role of RICIS then is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. RICIS, working jointly with its sponsors, advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research and integrates technical results into the goals of UHCL, NASA/JSC and industry.

Advanced Software Development Workstation Project Engineering Scripting Language Graphical Editor SUMMARY REPORT

RICIS Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Inference Corporation. Dr. Anthony Lekkos, Associate Professor, Computer and Information Sciences, served as RICIS research coordinator.

Funding was provided by the Information Technology Division, Information Systems Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Robert Savely of the Information Technology Division, Information Systems Directorate, NASA/JSC.

Table of Contents

List of Figures

1. Introduction

1.1 Motivation

Software development is widely considered to be a bottleneck in the development of complex systems, both in terms of development and in terms of maintenance of deployed systems. Cost of software development and maintenance can also be very high. One approach to reducing costs and relieving this bottleneck is increasing the reuse of software designs and software components. A method for achieving such reuse is a software parts composition system. Such a system consists of a language for modeling software parts and their interfaces, a catalog of existing parts, an editor for combining parts, and a code generator that takes a specification and generates code for that application in the target language. The Advanced Software Development Workstation is intended to be an expert system shell designed to provide the capabilities of a software parts composition system.

1.2 Project Background

The first phase of the Automated Software Development Workstation Project began in the fall of 1985 and work has continued to the present. The first phase demonstrated in a limited domain (Space Station momentum management) the feasibility of a knowledge-based approach to the development of a software components composition system. The second phase, which began in April, 1987, focused on ways to exploit knowledge representation, retrieval, and acquisition techniques to reduce the amount of effort required to build such systems. The third phase focused on enhancement of the prototype system developed in Phase II and addressed issues of scale-up and integration with present or future NASA software development environments. For Phase III work, emphasis has been on technology transfer to groups within NASA Johnson Space Center (JSC) and the NASA community and the use of the ASDW prototype in actual software configuration activities. In addition, work was been done on the generalization of the ASDW framework to support use of the system as a generic design knowledge acquisition system. ACCESS is the name of the prototype software for the ASDW. It is a knowledge-based software information system designed to assist the user in modifying or configuring retrieved software or design objects to satisfy user specifications.

Phase IV of this project consisted of two parts. The first effort was directed to providing additional enhancements to the ACCESS user interface, as requested by NASA JSC, based on feedback from the target community of users in Mission Operations.

The second, and more significant, effort consisted of extending the ACCESS framework to include a graphical user interface to support the specification of application modules

using the Engineering Scripting Language (ESL) developed by SofTech. This editor allows end-users of the ACCESS system to add to the library of application components without the need for a knowledge engineer as an intermediary, thus removing a significant roadblock to the wider adoption of the ACCESS technology within the target user community.

1.3 Status

The basis for the representing knowledge in ACCESS is ART-IM®, a toolkit for the development of knowledge-based expert systems. The ART-IM schema system is used as the mechanism for representing objects within ACCESS. ART-IM rules are used to propagate constraints within the object system and to test for constraint violations.

The user interacts with ACCESS via a graphical, point and click interface. This interface has been developed using the facilities of TAE Plus (Transportable Applications Environment Plus), which provides capabilities for developing interfaces on top of the X Window System. The user interface to ACCESS is designed to hide the details of the ART-IM language from the end user. Standard panels are provided for the user to browse and modify objects in the knowledge base. In addition, the knowledge engineer who develops a knowledge base to use with ACCESS can also use TAE Plus develop custom forms for browsing or modification.

The graphical interface which supports ESL was developed using InterViews and Unidraw. InterViews is a graphical user interface toolkit, developed at Stanford University and available in the public domain. Unidraw is a library built on top of InterViews which defines basic abstractions for building graphical editors.

The current prototype runs on Sun hardware, but as the source languages for ART-IM, TAE Plus, InterViews, Unidraw and ACCESS are C++ and C, ACCESS is readily portable to a wide variety of platforms.

2. Engineering Scripting Language

2.1 Overview

As part of an earlier study done for UHCL/RICIS, Softech partially specified a graphical language for building applications software - an Engineering Scripting Language (ESL). As described in the Summary Report on this project [Softech 90], "the purpose of an ESL is to allow the application engineer to limit his view of the problem space to those functional concepts that are inherent in the problem space and not in the software design."

The ESL described in this report is a graphical language for representing applications. As originally envisioned by Softech, each graph consists of nodes, which may be separate Ada tasks, and queues, which represent the flow of data from one node to another. Graphs formed in this way can then be treated as components for other graphs - such a graph is called a subgraph.

For the initial ESL editor prototype, a subset of the functionality described in the Summary Report was implemented. In this prototype, the node primitives represent Ada procedures or functions. In general, a connection between two nodes represents a single data item being passed from one node to another, rather than a queue of data objects. There are no explicit Merge or Replicator nodes - this functionality is supplied implicitly by allowing multiple connectors to or from a single data port. The prototype supports IF nodes and ITERATOR nodes, which translate to Ada `IF` and `LOOP` constructs. The ESL as implemented is described in more detail in the following sections.

2.2 Graph Objects

The ESL Editor supports the graphical construction of Ada programs through the use of SUBPROGRAM objects. A SUBPROGRAM object represents an Ada procedure and function. Such an object can be either a PRIMITIVE-SUBPROGRAM object provided by a knowledge engineer (KE-provided) or a GRAPH object built up from these primitive objects.

GRAPH objects are represented visually as a collection of nodes (boxes) connected by connectors (lines) with special boxes representing graph inputs and graph outputs. Each node in a graph is associated with either a PRIMITIVE-SUBPROGRAM object, another GRAPH object, or an Ada IF or LOOP construct. The connections from one node to another node represent data or control being passed. Nodes may be connected on either side; connections on the right side are input connections **to** the node, and connections on the left side are output connections **from** the node.

Additionally, the graph itself can have inputs and outputs, and the ESL graph must contain boxes representing graph inputs and graph outputs. The box or pseudo-node representing graph inputs only allows outgoing connections. Similarly, the box for the graph outputs only supports incoming connections.

Each SUBPROGRAM object has INPUT-PORTS and OUTPUT-PORTS. An input port corresponds to an Ada IN (or IN OUT) parameter; an output port corresponds to an Ada OUT (or IN OUT) parameter or a return value from an Ada function. Visually, the ports on a node are represented by two small rectangular areas on the left (for input ports) or right (for output ports) sides of the box representing the node.

Conceptually, between any two nodes, there can be a group of connectors - each connector representing a single data item or execution "trigger" passed from an output port of the first node to an input port of the second node. To support the concept of a trigger connection, each node is considered to have an implicit input port and an implicit output port of type trigger - named READY and DONE.

2.3 Graph Validation

One of the most significant technical challenges in this project was to clarify the concept of a "valid" graph - that is to develop a method for determining whether a graph constructed using the ESL Editor is syntactically correct. It was decided that in order for a graph to be valid, the following two conditions must hold:

- First, it must be possible to interpret the graph in such a way that a syntactically correct Ada program can be generated from the graph.

- Second, the graph must be constructed in such a way once the code is generated, data is computed for all graph output ports regardless of the execution path followed. This means, in the simplest case, that if a value which is a graph output is computed by following the THEN branch of an IF node, then it must also be computed when following the ELSE branch of that IF node.

In addition, certain limitations have been placed on the types of graphs which could be constructed based on feasibility of implementing an effective validation algorithm.

It is important to realize that the validation algorithm as developed cannot "prove" that a program generated from a valid graph is correct in the sense of producing meaningful results. For example, without additional restrictions, there is no way of checking that loops are exited, that is, that the boolean test at the end of a loop will at some point return FALSE.

2.3.1 Graph Semantics

This section describes various clarifications and interpretations made of graph semantics. A primary motivation for many of these interpretations is the goal of producing readable code from the graph.

Note that an ESL graph has an interpretation as a directed graph in which the nodes are vertices and the connectors between nodes are edges. This interpretation will be referred to in what follows.

1. If a node output port is not connected to an input port, then the output from that node simply isn't used. It is not necessary to explicitly force a connection to NULL.

2. It is possible to have multiple connectors with the same input port as destination - this amounts to an implied merge. Because of this interpretation, explicit merge nodes are not required and are supported in the initial ESL editor implementation.

3. It is possible to have multiple connectors with the same output port as source - this amounts to an implied replicator. Because of this interpretation, explicit replicator nodes are not required and are not be supported in the initial ESL implementation.

4. A special node type, the ITERATE node, will be used to support looping. The ITERATE node has two input ports of boolean type , the INITIAL port and the LOOP port. It has two output ports of trigger type, the CONTINUE port and the EXIT port. The INITIAL port of the ITERATE node must be connected to a port outside the loop; there must be a connection to the CONTINUE port from within the scope of the loop.

   The code which is generated for an ITERATOR node is as follows:

   ```
   while (<boolean-variable>) loop

   end loop;
   ```

   The value <boolean-variable> is set outside the loop to the value passed by a connector to the INITIAL port. Subsequently, its value is set within the loop to the value passed by the connector to the LOOP port.

   A node is within the scope of a loop if there is a simple graph cycle which contains the ITERATOR node (vertex) and the connectors (edges) from the CONTINUE port and into the LOOP port of that node.

   No node can correspond to code which is executed more than once except nodes within the scope of a loop. In other words, an IF or SELECT node cannot be used to support looping.

5. If an output port from one node is connected to the input port on another node, then it is possible that data will be available at the input port more than once before the second node is "ready" to be executed. This occurs when a value is computed multiple times within a loop, but used only after exiting from the loop. The interpretation in this case will be that the most recent value available at the port will be the value which will be used.

6. If a node is executed more than once (e.g., is within the scope of a loop), then it is possible that a value will be available at an input port the first time the node is executed and that no new input will be available at this node subsequently. To support this type of situation, the interpretation will be that a node input is not "consumed" until another input replaces it.

The graph validation algorithm is describe in more detail in Appendix A.

3. User Interface

The user creates and opens ESL Graph objects using the OPEN button on the ACCESS Tools Panel; to create a new ESL Graph, one selects GRAPH in the object taxonomy and then uses the OPEN button to create a new instance. Because the graph name is the name used for the Ada subprogram generated, the new object name must be a legal Ada identifier. Ada identifiers begin with a letter and then are followed by an arbitrary number of letters, numbers, and underscores.

When an ESL Graph object is opened the ESL Editor Panel and ESL Editor Control Panels are displayed. The ESL Editor Panel shows the image for the graph and contains the tools used to create, move, and connect the graph's nodes. The ESL Editor Control Panel has a list of the nodes on the graph and status information and contains the commands that change, and save the graph being viewed, commands to delete or change detailed information about nodes or connectors, and commands used to generate the Ada program corresponding to the graph. The ESL Editor Panel is shown in Figure 3-1 and the ESL Control Panel is shown in Figure 3-2.

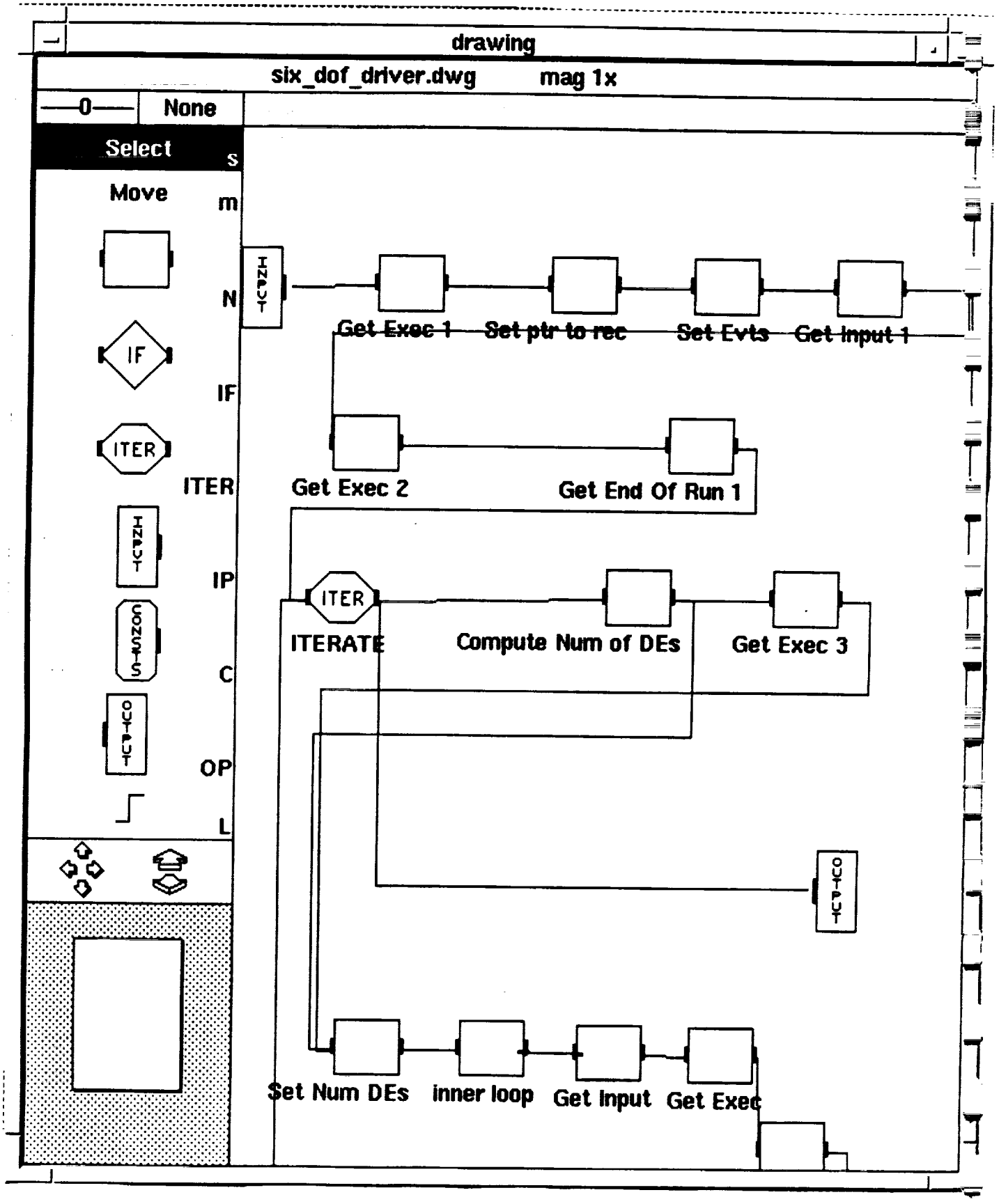
3.1 The ESL Editor Panel

The ESL Editor Panel is the primary editing panel for ESL Graphs. The panel is broken up into several parts. The top portion of the panel contains the magnification level and the name of the drawing file associated with the graph. Below the status line on the left are the Tools Palette and the panner. Below the status line on the right is the ESL Workspace where the graph is shown.

In the ESL Workspace, the nodes on the graph are shown as boxes with terminals on the left and right sides to which connections can be made. Connections on the graph are shown as lines emanating from the right side of one node, connecting to the left of another. Additionally there are three types of special "pseudo-node" for the graph's input ports, output ports, and for constant sources. Because the graph input ports and constant values can only be sources of connections, the corresponding pseudo-nodes only have a right side terminal, and similarly, the graph output ports pseudo-node only has a left side terminal.

To the left of the drawing workspace is a palette of tools, one of which is always selected. Clicking the first mouse button in the Graph Workspace wields the currently selected tool at the mouse location. To switch to another tool, one moves the mouse over that tool and clicks the first mouse button. As a short cut, one can click on the middle mouse button to wield the Move Tool or click on the third mouse button to wield the Select Tool. The system's response to wielding a tool depends on the type of tool wielded and the location of the mouse when the tool is wielded.

Figure 3-1: ESL Editor Panel



- **Select.** The topmost tool on the Tools Palette is the Select Tool. When this tool is wielded and the mouse is over a node or connector, that object is selected. That object is then the one affected by commands on the Edit Menu on the ESL Editor Control Panel. (The ESL Editor Control Panel is discussed in Section 3.2.

- **Move.** Below the Select Tool is the Move Tool. Nodes may be moved by wielding this tool over the node and moving the mouse while keeping the first mouse button selected.

- **Subprogram Node Creator.** Below the Move Tool is the Subprogram Node Creator Tool. If the currently selected object on the Tools Panel is a Subprogram object, then when this tool is wielded, a node corresponding to that Subprogram is created at the mouse position. Using this tool, the user may create either a Primitive Subprogram Node that represents some KE-provided primitive subprogram, or a Subgraph Node that represents a KE-created or a user-created ESL Graph.

  After the node is created, a Node Details Panel corresponding to the new node is displayed. The Node Details Panel is discussed in Section 3.3.

- **If Node Creator** and **Iterator Node Creator.** These two tools are used to create IF Nodes and ITERATOR Nodes respectively. If Nodes correspond to Ada IF statements and Iterator Nodes correspond to Ada LOOP statements. The use of these nodes and the ESL Graph language was discussed in chapter 2. After creating the new node, a Node Details Panel is displayed for the new node.

- **Graph Input Ports Creator, Constant Source Creator**, and **Graph Output Ports Creator.** These tools make the "pseudo-nodes" that stand for the graph's input ports, for constant sources, and for the graph's output ports respectively. A graph can have at most one of each type of these pseudo-nodes; to be valid, a graph must have the Graph Input Ports and Graph Output Ports pseudo-nodes.

  As discussed earlier, connections can only be made **from** the Graph Input Ports and the Constant Source pseudo-nodes, and likewise, connections can only be made **to** the Graph Output Ports pseudo-node.

- **Connection Tool.** Below the node creation tools on the Tools Palette is the Connection Tool. This is used to create the connections between the nodes. To create a connection, the user clicks the **first** mouse button when the mouse is on the terminal that is to be the source of the connection. A connecting line will follow the mouse movement, but only horizontally and

vertically. One can effect a 90 degree turn in the connection by clicking on the first mouse button. To terminate the connection, one click with the **second** mouse button when the the mouse is over the destination terminal.

After the connector has been made, a Connector Details Panel is displayed for the new connector. With this panel the user may specify which ports that are to be connected between the two nodes. The Connector Details Panel is discussed in Section 3.7.

Below the tools is a "panner" that allows the user to shift the graph workspace view. The four direction arrows shift the workspace up, down, left, and right, and the in and out arrows cause the view to zoom in and out. Below the arrows is a part of the panner that represents the ESL Workspace. The view may be shifted up, down, left, or right by dragging the solid white portion, or by clicking in the grey area.

The ESL Editor Panel cannot be resized.

3.2 The ESL Editor Control Panel

The ESL Editor Control Panel contains three menus - the View Menu, the Edit Menu, and the Translate Menu. Below the menus are fields indicating the graph being edited, the root graph (the graph that was first opened), and the parent graph of the current graph. Below these fields is a field that indicates the object currently selected on the ESL Workspace. It is this object that the functions on the Edit Menu operate on. Beside these fields is a field containing a list of the nodes on the graph. Selecting an item on this list selects the corresponding node in the ESL Workspace and vice-versa.

- **View**. The View Menu contains commands that affect the graph being viewed. The commands on this menu are Delete, Save, Parent Graph, Subgraph, and Close.

  Delete deletes the graph currently being edited. Once a graph is used in another graph, it cannot be deleted. Graphs must be deleted from this menu. The Delete command in the Object Menu on the Tools Panel does not work on Graph objects.

  The Save command saves changes made to the current graph. If the current graph is modified, the graph must be either saved or deleted before the ESL Editor may be closed or another graph opened.

  If the object currently selected in the ESL Editor Panel is a Subgraph node, then the Subgraph function makes the ESL Editor Panels show that child graph. The root graph stays the same, but the parent graph is the graph that was previously being viewed. The Parent Graph function switches the view back to the parent graph.

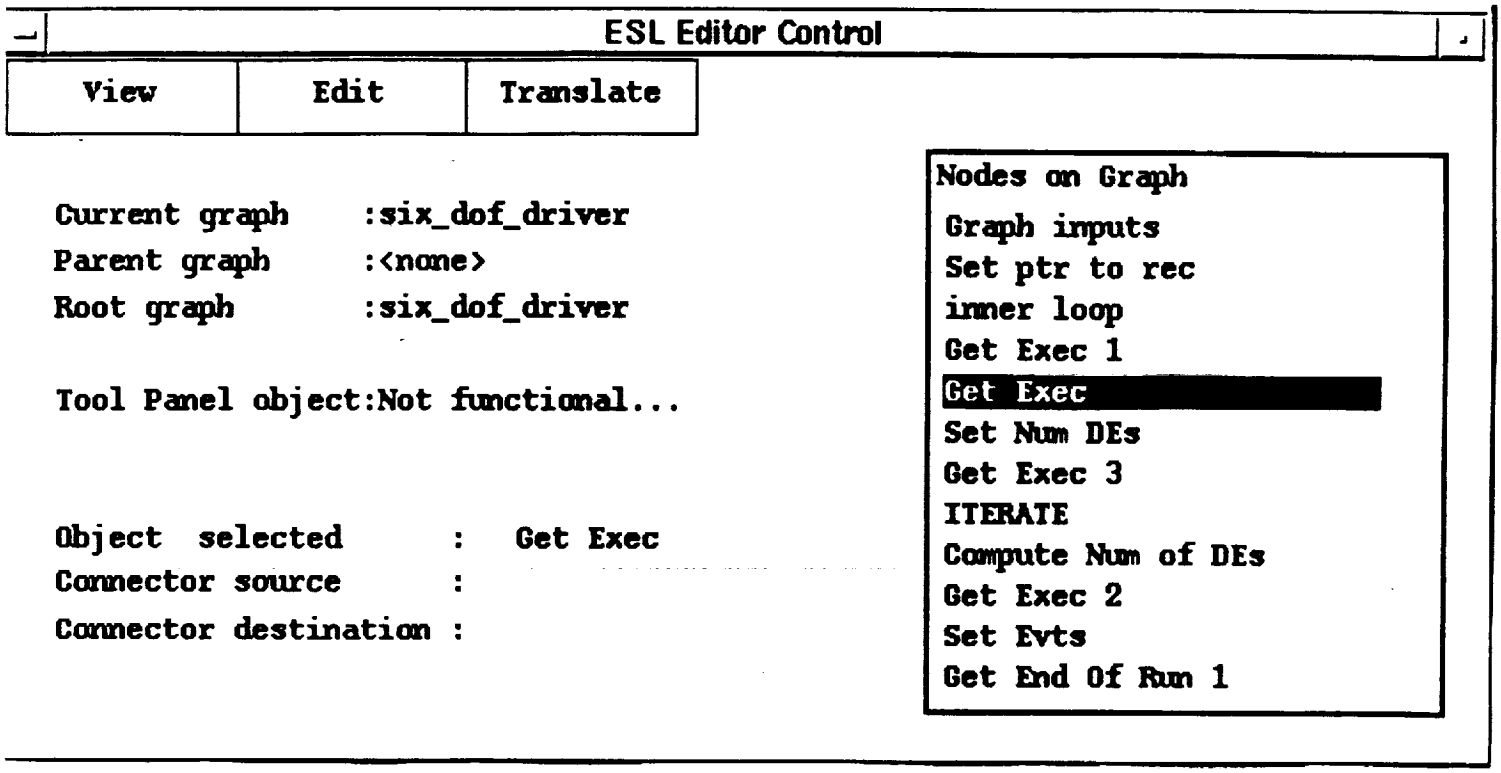


Figure 3-2: ESL Editor Control Panel

The Close command dismisses both the ESL Editor Panel and the ESL Editor Control Panel. If the graph has been modified, it must be either saved or deleted before the ESL Panels may be closed.

- **Edit.** The Edit Menu contains commands that change or display information about the object currently selected in the ESL Workspace. It contains two commands, Delete and Object Details...

  Delete deletes the node or connector that is selected on the ESL Workspace. When a node is deleted, all connectors to that node are also deleted.

  Object Details... displays a panel giving detailed information about the selected object. If a node is selected, then a Node Details Panel for that node is displayed. If a connector is selected, then a Connector Details Panel for that connector is displayed. The Node Details Panel is discussed in Section 3.3 and the Connector Details Panel is discussed in Section 3.7.

- **Translate.** The Translate Menu contains commands related to generating the Ada code for a graph. It contains three functions - Validate Current Graph, Validate Entire Graph, and Generate Ada. Graph validation is discussed in Section 2.3.

The Validate Current Graph command validates only the graph currently being viewed. The Validate Entire Graph command validates the current graph and all subgraphs below it.

Generate Ada first validates the current graph and all its subgraphs, then generates the corresponding Ada programs. If the code for the graph has already been generated since the last modification to the graph and the generated file still exists, then no code generation is performed. The filename for the code generated for a graph is the graph name followed by ".a".

3.3 The Node Details Panel

The Node Details Panel is used to view information about a node on a graph. From this panel the user may also change the name of the node or pop up a Notes Panel where notes about the node's use may be browsed or edited. The Node Details Panel is shown in Figure 3-3. The contents of the fields on the panel and responses to events are described below:

- **Name.** The Name Field is a text field which contains the node's name. Initially a node's name is system-generated. This field is read and the node's name is changed when the user terminates input to the field.

- **Type.** The Type Field identifies the type of the node. Examples of values displayed in this field are "Procedure," "Function," "Subgraph," and "If." The text in this field is read only.

- **Input Ports and Output Ports.** The Input Ports and Output Ports Fields are textlist fields which list the input ports and output ports for the node and the connections they are associated with. Items in these fields give the port name, data type, and a connection status string.

  The current selection of either textlist is read when the Connector Details... Button immediately below the textlist is selected.

- **Connector Details...** Below both the Input Ports and Output Ports textlist fields is a Connector Details... button. When this button is selected, a Connector Details Panel is popped up for the connector group that corresponds to the current selection in the textlist. The Connector Details Panels are discussed in Sections 3.7, 3.8, and 3.9.

- **Component Details...** When the Component Details... Button is selected, a Component Details Panel is popped up for the subprogram corresponding to the node. The Component Details Panel is discussed in Section 3.4.

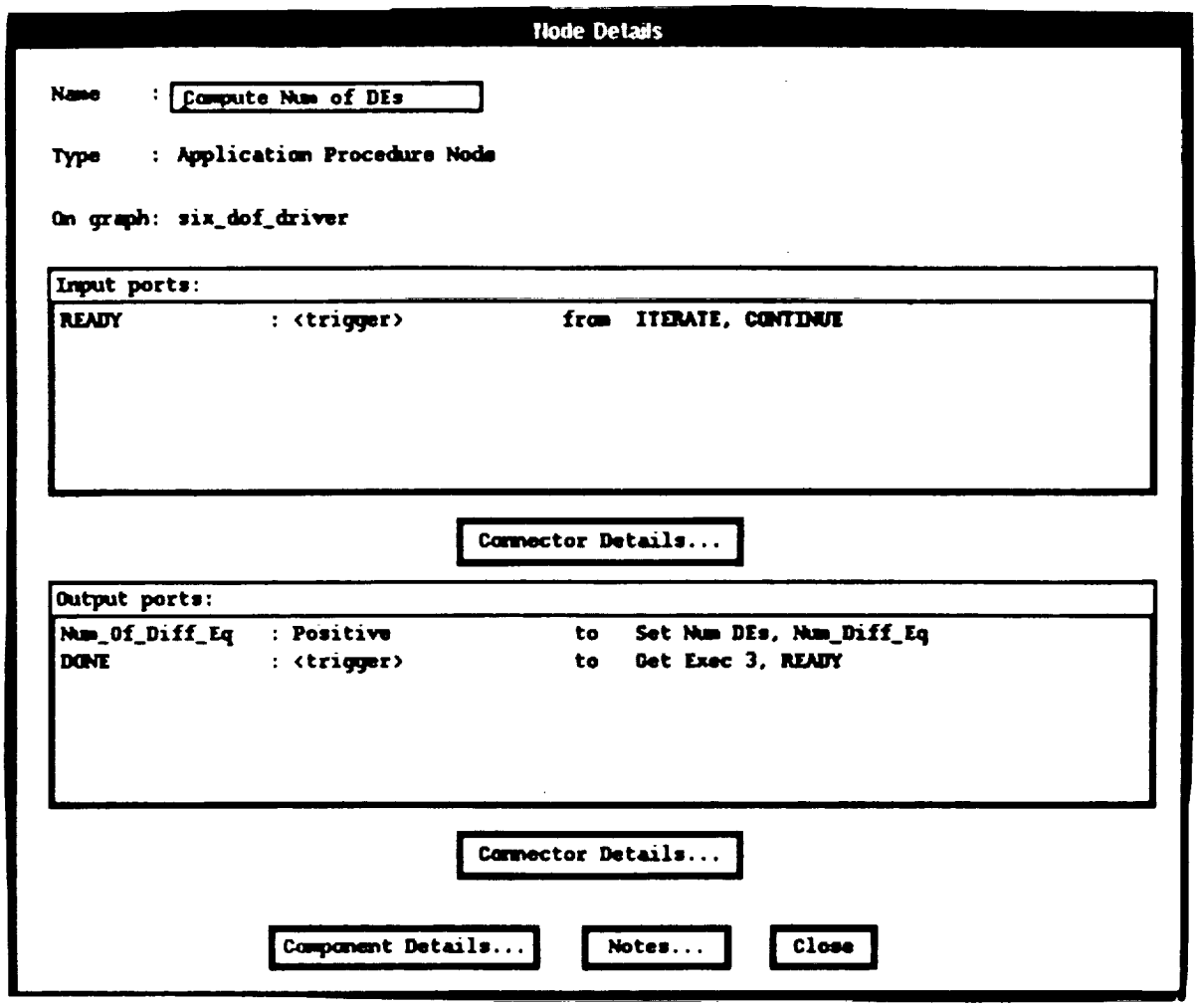


Figure 3-3: Node Details Panel

- **Notes...** When the Notes... Button is selected, a Node Notes Panel is popped up.

  The Node Notes Panel contains fields that identify the node name and type, a field with Component comments, a field with user-editable notes about the node, an Ok button, and a Cancel button. When the Ok button is selected, changes to the node's notes are made permanent and the panel is dismissed. The Cancel button dismisses the panel without making any changes.

- **Close.** When the Close Button is selected, the Node Details Panel is dismissed.

3.4 Component Details Panel

The Component Details Panel is used to display information about a program component (procedure, function, or subgraph) which can be used as a node on another graph. The Component Details Panel is shown in Figure 3-4. The following describes the fields on this panel:

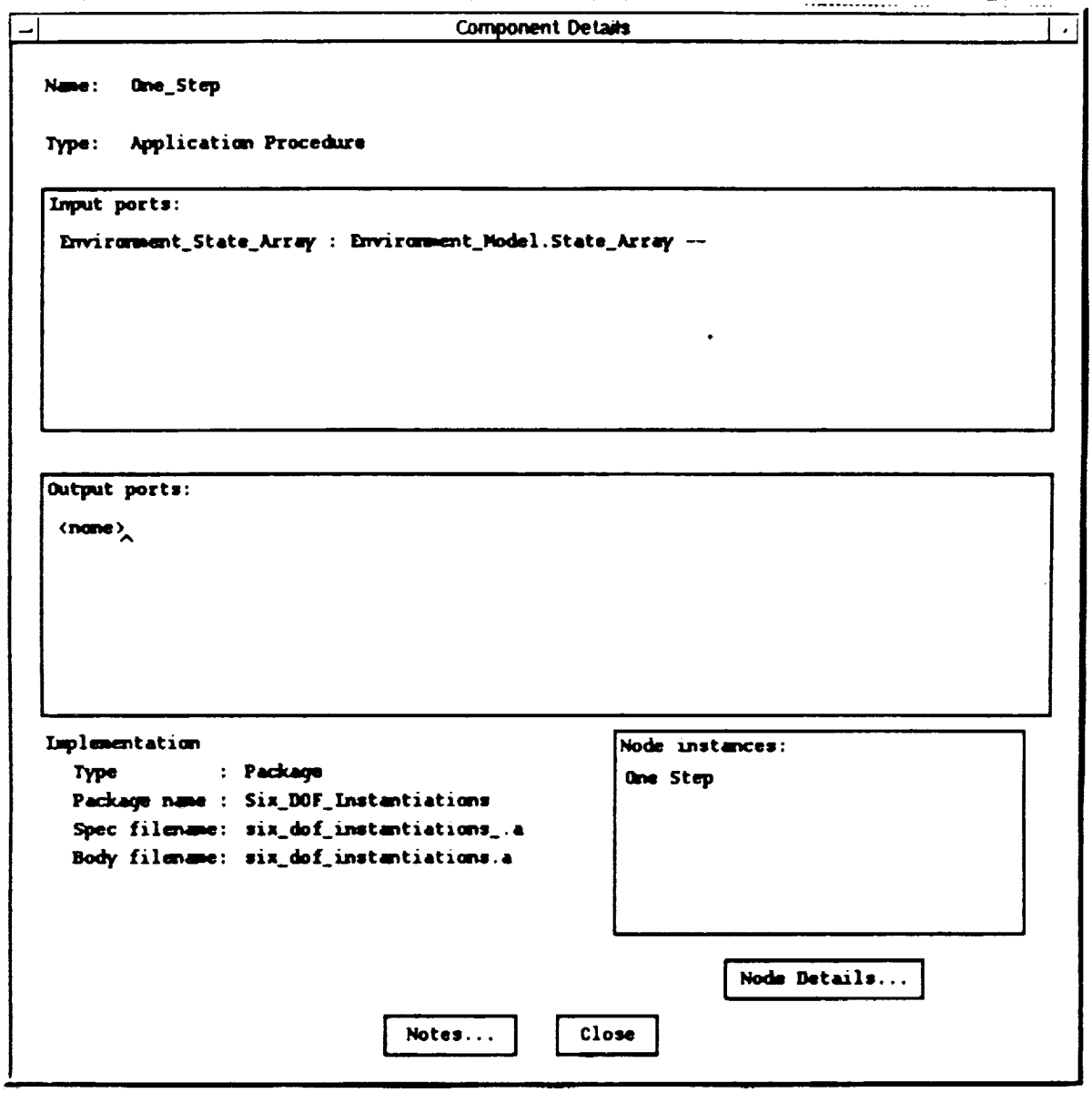


Figure 3-4: Component Details Panel

- **Name** and **Type.** The Name and Type Fields identify the component name and type. Component types are either Procedure, Function, or Graph. The text in these fields is read only.

- **Input Ports and Output Ports.** The Input Ports and Output Ports

Fields list information about the component's input and output ports. Both fields are populated with lines that show the port name, data type, and port comments. For procedure or function components, the port comments are supplied by a Knowledge Engineer. For Graph components, the port comments are the same as the notes for the connector to the graph port. The text in these fields is read only.

- **Implementation** fields. The details about a component's implementation are listed in four implementation fields. The text in all these fields is read only. The fields are:
    - **Type.** This field contains a string indicating whether implementation type is Inline, Procedure, or Package.
    - **Package Name.** If the Implementation Type is Package, then this field contains the package name.
    - **Spec Filename.** If the Implementation Type is Package, then the filename for the Ada package spec is displayed in this field.
    - **Body Filename.** If the Implementation Type is Package or Procedure, then the filename for the Ada package or procedure body is displayed in this field.

  For Procedure or Function Components, the implementation information is contained in objects created by the Knowledge Engineer. For Graph Components, the implementation information is entered by the user.

- **Node Instances.** The Node Instances Field is a textlist field which contains the names of all nodes which use this program component. The current selection from this field read when the Node Details... Button is selected.

- **Node Details...** When the Node Details... button is selected, a Node Details Panel is popped up with information about the node corresponding to the currently selected item in the Node Instances textlist. The Node Details Panel is discussed in Section 3.3

- **Notes...** When the Notes... Button is selected, a Component Notes Panel is popped up.

  The Components Notes Panel contains static text fields displaying the component name and type, a non-editable field of comments about this component, and a Close button. Selecting the Close Button dismisses the panel.

- **Close.** When the Close Button is selected, the Component Details Panel is dismissed.

3.5 Graph Port Details Panel

The Graph Port Details Panel is used to browse the set of input or output ports for a graph. This panel is popped up after the user has selected either the icon representing the current graph's input ports or the icon representing the current graph's output ports and then selected the Object Details... menu item on the Edit Menu. This panel is shown in Figure 3-5.

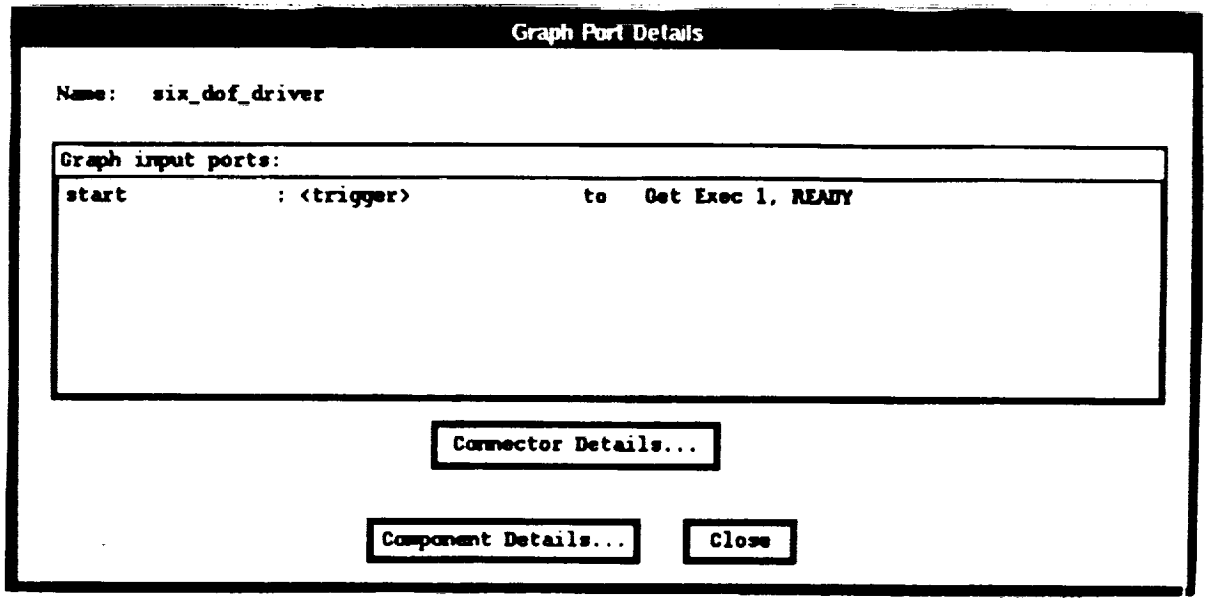


Figure 3-5: Graph Ports Details Panel

The fields on the Graph Port Details Panel are as follows:

- **Name.** The Name Field displays the graph name. The text in this field is read only.

- **Graph Ports.** The Graph Ports Field is a textlist field which lists information about the graph ports being examined. The title of the textlist identifies whether the panel is displaying information about the graph's input ports or its output ports. An item in the Graph Ports textlist gives the name of graph port, its data type, and its connection status.

  The current selection from this field is read when the Connector Details... button is selected.

- **Connector Details...** When the Connector Details... Button is selected, a Connector Details Panel is popped up and displays information about the

connector group corresponding to the currently selected item on the Graph Ports textlist. The Connector Details Panels are discussed in Section 3.7, 3.8, and 3.9.

- **Component Details...** When the Component Details... Button is selected, a Component Details Panel for the graph to which the graph ports belong is displayed. The Component Details Panel is discussed in Section 3.4.

- **Close.** When the Close Button is selected, the Graph Port Details Panel is dismissed.

3.6 The Connector Details Panels

The Connector Details Panels are used to create, modify, or examine the connector groups on a graph. All connectors in a connector group have the same Source and Destination, and all connectors that have the same Source and Destination are in the same group. The Source of a connector can either be a Node, a Constant value, or a Graph Input Port. The Destination of a connector can either be a Node or a Graph Output Port. Depending on the Source and Destination of the connector group, one of the following four Connector Details Panels will be popped up:

- Node to Node Connector Details
- Constant to Node Connector Details
- Graph Input Port to Node Connector Details
- Node to Graph Output Port Connector Details

Since the Node to Node Connector Details is the most general of the Connector Details Panels, it will be discussed first. Then the others will be discussed in turn. The Node to Node Connector Details Panel and the Node to Graph Output Ports Connector Panel are shown in Figures 3-6 and 3-7 respectively.

3.7 Node to Node Connector Details Panel

The Node to Node Connector Details Panel is used when both the Source and Destination of the connector group are nodes. The Node to Node Connector Details Panel is shown in Figure 3-6. The contents of fields on the panel and responses to events are described below:

- **On Graph.** The On Graph Field contains the name of the graph that the connector belongs to. The text in this field is read only.

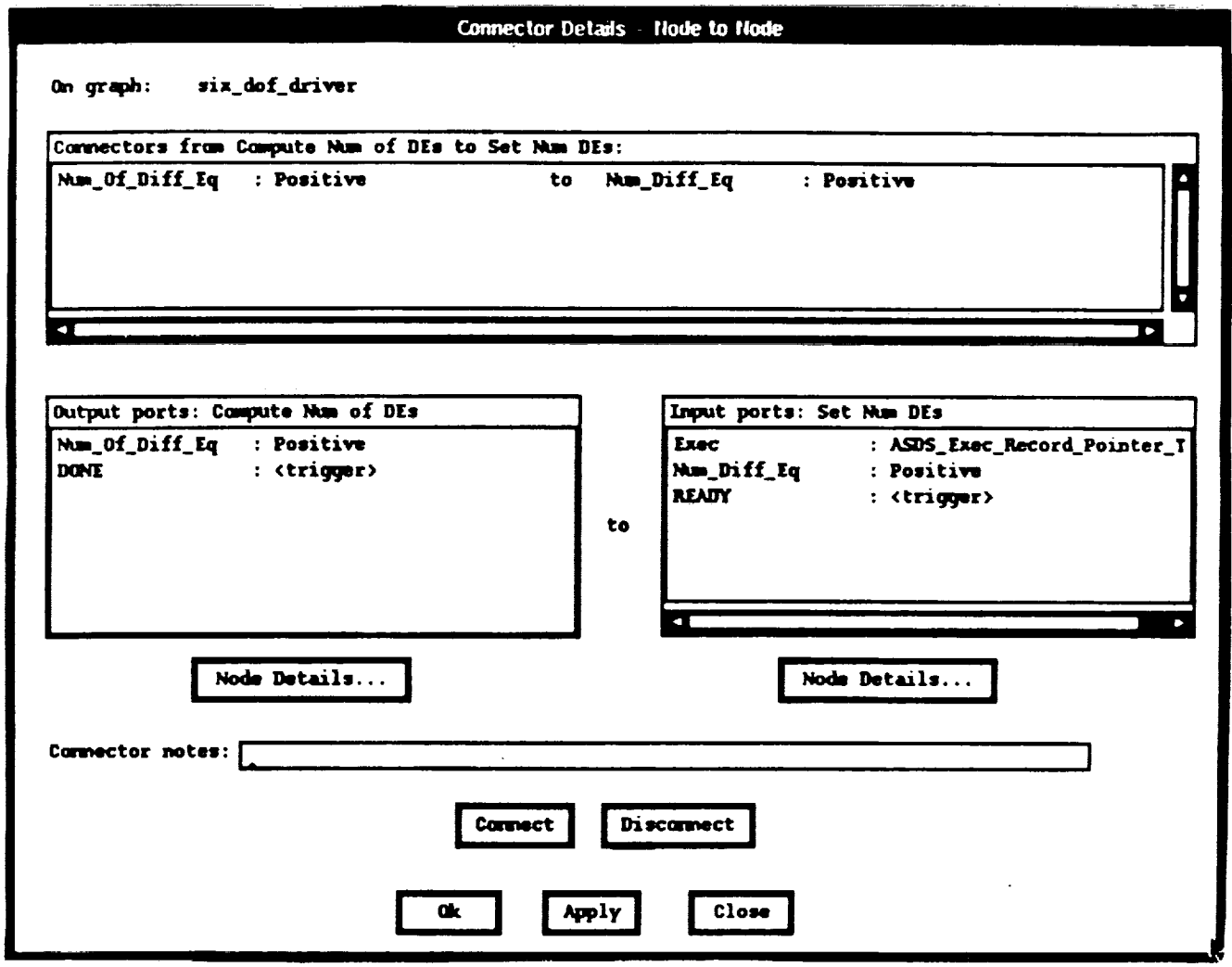


Figure 3-6: Node to Node Connector Details Panel

- **Connectors.** The Connectors Field is a textlist field which lists all the connections between the Source and Destination. The title of the textlist identifies the Source and Destination of the connector group. The contents of the textlist are items with fields for the source port name and data type, and the destination port name and data type. The source and destination ports always have the same data type. If there are no connectors between the Source and Destination, then the textlist contains the single item "<none>", which, if selected, becomes unselected immediately.

  Selecting an item on the Connectors textlist causes the selection of the items in the Sources textlist and Destinations textlist that correspond to the source and destination of the selected connector item. The Connector Notes text field will be populated with user-entered notes about this connector.

  The current selection on the Connectors textlist is read when the Disconnect Button is selected. The Disconnect Button is described below.

- **Source Ports and Destination Ports.** The Source Ports Field is a textlist field listing the output ports of the Source node. Its title identifies the Source node. Similarly, the Destination Ports Field lists the input ports of the Destination node and its title identifies the Destination node. Both fields are populated with items that include the port name and data type.

  When an item on the Source Ports textlist or Destinations Ports textlist is selected, one of two things will happen. If the item selected corresponds to a port that is already the source or destination of a connector between the two nodes, then the corresponding connector item in the Connectors textlist is selected, as is the corresponding source or destination port item on the other textlist. The Connector Notes text field is populated with any user-entered notes about the connector.

  If the item selected does not correspond to a port that is a source or destination of a connector between the two nodes, then any selected item in the Connections textlist is unselected and any corresponding text in the Connector Notes Field is also cleared. If there is an item selected in the other (Source Ports or Destination Ports) textlist and it is not of the same data type as the item just selected, then it is unselected as well.

  All nodes have an output port with trigger data type named "<done>". It is triggered after the node has executed. Likewise, all nodes have an input port with trigger data type named "<ready>". When HOL code is generated from a graph, it will be generated in such a way that the code corresponding to a node with a trigger connection to second node is executed prior to the code corresponding to the second node. This is described in more detail in Chapter 2.

  The current selections on the Source Ports textlist and Destination Ports textlist are read when the Connect button is selected.

- **Node Details...** Below the Source Ports textlist is a button labeled Node Details... When it is selected, a Node Details Panel with information about the Source node will be popped up. A similar button appears below Destination Ports textlist. The Node Details Panel is discussed in Section 3.3.

- **Connector Notes.** The Connector Notes Field is a text field containing one line of user-editable notes about a specific connection. When an item is selected on the Connections textlist, the Connector Notes Field is populated with the notes for that connector. If an item is selected in the Connectors textlist, the Connector Notes Field is read when the user terminates input to that field.

If no item is selected in the Connectors textlist, then the Connection Notes field can be used to enter notes about a new connector. When this is the case, the Connector Notes field is read when the Connect button is selected.

- **Connect.** The Connect Button is used to define a connection between a port on the Source node and a port on the Destination node. When selected, a new connection is defined between the port corresponding to the currently selected item on the Source Ports textlist and the port corresponding to the currently selected item on the Destination Ports textlist. An item for the new connector object is added to the Connectors textlist and is selected. The Connector Notes field is read and its contents are now associated with the new connection definition. The definition is not transformed into an actual connector object until the Apply or Ok Button is selected. A definition which has not been applied is discarded if the panel is dismissed with the Close Button.

- **Disconnect Button.** The Disconnect Button is used to specify deletion of an existing connector object. When selected, the currently selected item on the Connections textlist is deleted and the Connectors textlist is left with no item selected. Any selected items on the Source Ports textlist or on the Destination Ports textlist are unselected. The Connector Notes field is cleared. The corresponding connector object is not permanently deleted until the Apply or Ok Button is selected.

  If there are no more connectors between the Source and Destination, then the Connectors textlist is populated with the single item, "<none>".

- **Apply.** When the Apply Button is selected, any connection definitions or deletions entered through the Connector Details Panel are transformed into actual connector objects or deletions of connector objects. If any constraint violations have occurred, a warning panel is displayed.

- **Close.** When selected, the Close button dismisses the panel without propagating any changes entered since the last apply.

- **Ok.** Selecting the Ok button is equivalent to selecting Apply followed by Close.

3.8 Constants to Node Connector Details Panel

The Constants to Node Connector Details Panel is used when the source ports for a connector group are all constants. The fields on this panel are the same as those on the Node to Node Connector Details Panel, with one exception. Instead of a Sources textlist, a text field is provided to allow the user to enter a constant value. Below this

Constant Value Field is a button which when selected will cause a panel to pop up to help the user choose a constant value.

The differences in behavior between the Constant to Node Connector Details Panel panel and the Node to Node Connector Details Panel are listed below:

- **Connectors.** Items in the Connectors field do not contain explicit information about the data type of the connector's source. This allows more room for the constant value. When an item is selected, the Constant Value Field is populated with the value of the Source Port of the corresponding connector.

- **Constant Value.** The Constant Value Field is a text field which is used to enter a constant value to be used as the source of a connector. If the value of this field is changed while an item on the Connectors textlist is selected, its value is tested for validity. This means that it is tested to see if it is a valid identifier for the data type of the input port to which it is intended to be connected.

  A description of how identifiers are tested for validity is contained in Section 5.3.

- **Select...** The Select... Button is used to help a user select a constant value to be applied to a given port. To be used, an item must be selected on the Destination Ports textlist, and a set of constant values for that data type must be defined. If this is the case, then the Select Constant Value Panel is popped up.

  The Select Constant Value Panel has a static text field that indicates the data type of the destination, a textlist that contains the defined constant values for that type, an Ok button, and a Cancel button. When an item is selected on the textlist and the Ok button is selected, the panel is dismissed and the Constant Value Field is populated with the selected value. The Cancel button dismisses the Select Constant Value Panel and leaves the Constant Value Field unchanged.

- **Connect.** Before defining a connection, the constant source value is checked to see if it is a valid identifier for the data type of the destination port. If it is not, a warning panel is displayed and the connection is not defined.

- **Disconnect.** In addition to the actions described in Section 3.7, selecting Disconnect also clears the Constant Value Field.

3.9 Graph Input Ports to Node Connector Details Panel

The Graph Input Ports to Node Connector Details Panel is used to connect graph input ports to a node's input ports. Instead of a Source Ports textlist, it has a text field for entering a name for a graph input port. This text field behaves like the Constant Value Field on the Constants to Node Connector Details Panel. The panel has a Graph Port Details... button instead of a Node Details... button. When the Graph Port Details... button is selected, a Graph Port Details Panel is popped up for the graph's input ports. The Graph Port Details Panel is discussed in Section 3.5.

The Node to Graph Output Ports Connector Details Panel is used to connect a node's ports to graph output ports. It behaves analogously to the Graph Input Ports to Node Connector Details Panel. The Node to Graph Output Port Connector Details Panel is shown in Figure 3-7.

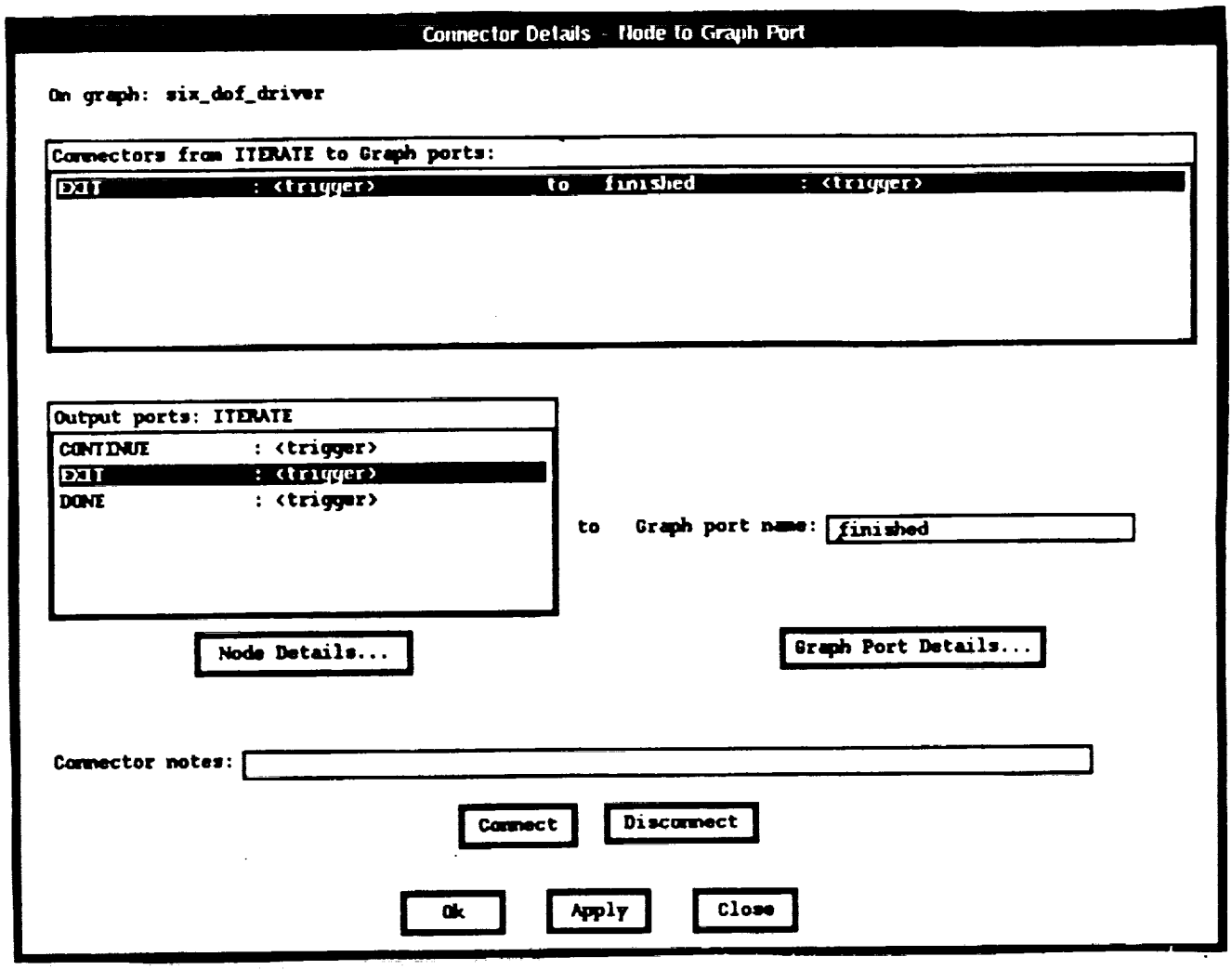


Figure 3-7: Node to Graph Output Ports Connector Details Panel

When defining a connection to a graph port that is already connected to another port, the data types must be compatible. If this is not the case, a warning panel is popped up and the connection definition is not made.

4. ESL System Architecture

The ESL software uses several different software libraries to support graphical editing of programs. The objects manipulated by the system are represented internally by ART-IM [Inference 91] schemas, the ESL Editor Panel is implemented using InterViews [InterViews 91], and the remaining ESL panels are implemented using TAE+ [TAE Plus 91]. The chapter briefly describes how each of these components fits into the ESL system architecture.

4.1 Use of ART-IM objects

ART-IM schemas are used to represent the objects manipulated by the ESL editor. Chapter 5 describes the structure for the schemas that the knowledge engineer must supply to the Access user. These KE-supplied objects describe the structure of the reusable components in a software library. In addition to these schemas representing the components used in graphs, the ESL system creates a schema corresponding to each user-created graph and node, as well as one for each of the connections made in a graph. In addition, as in earlier releases of ACCESS, the system maintains a schema for each distinct type of TAE+ panel which reflects the state of that panel.

4.2 Use of InterViews and TAE+

The ESL Editor Panel is implemented using functions in the InterViews and Unidraw libraries. InterViews and Unidraw are toolkits used to create graphical object editors. The software for the ESL Editor Panel is based on example programs supplied with Unidraw.

Creating an application that supports both the InterViews ESL Editor Panel and the various TAE+ panels required modifications be made to source code in both the InterViews and TAE+ software libraries. The InterViews software was modified so that the variable for the X windows display is shared between InterViews and TAE+ and so that initialization of that variable is done from TAE+.

The TAE+ functions also were modified so that they do not declare variables for the standard C++ streams, but rather share the ones declared and used by the InterViews software.

Finally, the TAE+ event handling loop was modified so that it detects when an event occurs in the InterViews panel. When this is the case, the event is placed back on the event queue and the InterViews function that handles the event is called. If not, the event is a TAE+ event and is handled as before.

5. KE Provided Objects

In order to create applications with the ESL Editor, the knowledge engineer must provide the user with a library of Ada subprograms and make these available to the ACCESS user by defining the corresponding ACCESS objects (ART-IM schemas).

This chapter describes the structure of these schemas. The file "esl-objects.art" contains a complete listing of the ESL schema definitions.

5.1 PRIMITIVE-SUBPROGRAM Objects

The knowledge engineer must define an instance of a PRIMITIVE-SUBPROGRAM schema for each Ada procedure or function that the user is to be able to incorporate into ESL Graphs. The structure of a PRIMITIVE-SUBPROGRAM schema is:

```
(defschema primitive-subprogram
           (label)
           (notes)
           (has-implementation)
           (has-input-ports)
           (has-output-ports)
           (subprogram-type)
  )
```

- LABEL. The value in the LABEL slot is the name of the Ada subprogram.

- NOTES. The value in the NOTES slot is a documentation string which is displayed in the Notes Field in the Component Notes Panel.

- HAS-IMPLEMENTATION. The value in the HAS-IMPLEMENTATION slot is the name of the IMPLEMENTATION schema for the node. The structure of IMPLEMENTATION schema is discussed in Section 5.4.

- HAS-INPUT-PORTS and HAS-OUTPUT-PORTS. The values in the HAS-INPUT-PORTS and HAS-OUTPUT-PORTS slots are the names of the PORT schemas for the subprogram. Both slots may have multiple values.

  There must be a PORT schema for each IN parameter, for each OUT parameter, and for the return value of a function. There should be two PORT schemas for IN OUT parameters. The structure of the PORT schema is discussed in Section 5.2.

- SUBPROGRAM-TYPE. The value in the SUBPROGRAM-TYPE slot is either PROCEDURE or FUNCTION, depending on the type of Ada subprogram.

5.2 PRIMITIVE-SUBPROGRAM-PORT Objects

The knowledge engineer must define an instance of a PRIMITIVE-SUBPROGRAM-PORT schema for each IN or OUT parameter or function return value for every Ada subprogram for which there is a corresponding PRIMITIVE-SUBPROGRAM object. The values in the HAS-INPUT-PORTS and HAS-OUTPUT-PORTS slots for this PRIMITIVE-SUBPROGRAM object are the names of these schemas. The structure of a PRIMITIVE-SUBPROGRAM-PORT schema is:

```
(defschema port
            (label)
            (notes)
            (direction)
            (port-data-type)
            (position)
            (parameter-type)
            (belongs-to-subprogram)
)
```

- LABEL. The value in the LABEL slot is the name for the port. The values in the LABEL slot of the two ports that correspond to an IN OUT parameter must have the same name.

- NOTES. The value in the NOTES slot is the documentation string for the port. This information appears after the port name in the Input Ports Field or Output Ports Field on the Component Details Panel. The Component Details Panel is discussed in Section 3.4.

- DIRECTION. The value in the DIRECTION slot is either INPUT or OUTPUT, depending on the port type.

- PORT-DATA-TYPE. The value in the PORT-DATA-TYPE slot is the name of the DATA-TYPE schema for the port. The structure of the DATA-TYPE schema is discussed in Section 5.3.

- POSITION. The value in the POSITION slot is the position of the parameter in the function calling sequence. The port for a function return value must always be first (POSITION 1).

- PARAMETER-TYPE. The value in the PARAMETER-TYPE slot is either IN, OUT, IN-OUT, or RETURN-VALUE, depending on the type of argument the port is used to represent.

- BELONGS-TO-SUBPROGRAM. The value in the BELONGS-TO-SUBPROGRAM slot is the name of the PRIMITIVE-SUBPROGRAM schema that the PRIMITIVE-SUBPROGRAM-PORT object corresponds to.

5.3 DATA-TYPE Objects

The knowledge engineer must define an instance of the DATA-TYPE schema for each type of data that is to be used by PRIMITIVE-SUBPROGRAM-PORT schemas and is named as a value in the PORT-DATA-TYPE slot of a PRIMITIVE-SUBPROGRAM-PORT schema. The PRIMITIVE-SUBPROGRAM-PORT is described in Section 5.2.

The structure of the DATA-TYPE schema is:

```
(defschema data-type
           (name-of-data-type)
           (defined-values)
           (test-function)
   )
```

- NAME-OF-DATA-TYPE. The value in the NAME-OF-DATA-TYPE slot is the name of the Ada data type. If the data type is defined in an Ada package, the "<package-name>.<data-type>" format should be used.

- DEFINED-VALUES. The values in the DEFINED-VALUES slot are names of defined constants for the data type. It is these values that the user chooses between when the user hits the Select Constant Value Button on the Constant to Node Connector Details Panel.

- TEST-FUNCTION. The value in the TEST-FUNCTION slot is the name of a function whose single argument is a string representing a user-entered constant. The function must return T or NIL depending on whether the value of the string is or is not valid for the corresponding data type.

5.4 IMPLEMENTATION Objects

The knowledge must define an IMPLEMENTATION schema for each subprogram object. This schema is the value of the HAS-IMPLEMENTATION slot of the corresponding SUBPROGRAM schema. If the Ada subprogram described by the SUBPROGRAM object is implemented as a separately compiled procedure, then the IMPLEMENTATION schema must be an instance of a SEPARATELY-COMPILED-PROCEDURE schema. If the subprogram is implemented as a visible procedures in a Ada package, then the IMPLEMENTATION schema must be an instance of a PACKAGE-IMPLEMENTATION schema. This IMPLEMENTATION schema may be referenced by all SUBPROGRAM objects corresponding to subprograms in the same package.

The structure of the variousIMPLEMENTATION schemas are as follows:

```
(defschema implementation)
```

```
(defschema separately-compiled-procedure
        (is-a implementation)
        (additional-withed-objects)
        (source-file-name)
        (object-file-name)
        (library-file-name)
)

(defschema package-implementation
        (is-a implementation)
        (name-of-package)
        (additional-withed-objects)
        (has-procedures)
        (package-spec-file-name)
        (package-body-file-name)
        (package-spec-object-file-name)
        (package-body-object-file-name)
)
```

- NAME-OF-PACKAGE. The value in the NAME-OF-PACKAGE slot is a string specifying the name of the corresponding the Ada package. When code is generated for a graph which contains a reference to a PACKAGE-IMPLEMENTATION object, An Ada "with" statement is generated for this package unless the string contains a ".". This string is also used in procedure calls to the subprogram objects implemented in this package. This slot only exists on PACKAGE-IMPLEMENTATION schemas.

  ADDITIONAL-WITHED-OBJECTS. The value in the ADDITIONAL-WITHED-OBJECTS is either a single string or a sequence of strings that list Ada packages or subprograms for which a "with" statement must be generated in order to be able to call the subprogram. Examples of such an object would be a package containing common data type definitions.

  If the value in the slot is a sequence of strings, then the order of these strings in the sequence determines the order that the Ada "with" statements are generated. Both types of IMPLEMENTATION schema have this slot.

- SOURCE-FILE-NAME. This slot is currently not used in KE-provided IMPLEMENTATION schema. In system-generated IMPLEMENTATION schemas for user-constructed GRAPH objects a string specifying the filename for the generated code is contained in this slot.

- HAS_PROCEDURES. This slot is not currently used.

- OBJECT-FILE-NAME, LIBRARY-FILE-NAME, PACKAGE-SPEC-FILE-NAME,

`PACKAGE-BODY-FILE-NAME`, `PACKAGE-SPEC-OBJECT-FILE-NAME`, and `PACKAGE-BODY-OBJECT-FILE-NAME`. These slots are not currently used.

6. Future Directions

There are two distinct directions in which additional work on the ESL system might be directed. There are also a number of possible enhancements to the system - both to expand the scope of the software generated by the system and to make the system easier to use. In addition, more work needs to be done in the design and engineering of reusable software libraries suitable for use with the ESL system.

6.1 Enhancements to the ESL System

There are several enhancements that could be made to the ESL system that would both improve the quality of the software generated by the system and also make the system more powerful and easier to use.

A limitation of the current ESL system is that it does not yet support asynchronous nodes, as was suggested in the ESL language specification. In the current implementation of the ESL language, nodes on the directed graph represent Ada procedures. The graph is used to generate an Ada main program, which can then be compiled and executed. The graph corresponds to a standard procedural programming paradigm with a predetermined flow of control.

An extension to this system would be for nodes to represent asynchronous processes executing in parallel. In that case, the graphical connections between these nodes would not represent a single data item, but rather a queue of data items. During execution of a node, data items would be consumed from the queue connected to each of the node's input ports and data items would be generated and put into the queue connected to a node output port. In this paradigm, a node is executed when there is sufficient data in each of the queues connected to the node's input ports. Such asynchronous nodes would probably be implemented as separate Ada tasks. Introduction of these asynchronous nodes will require a review of graph semantics, and will introduce changes to the graph validation and code generation processes.

The existing ESL system is limited in its support of Ada generics. Ada generics are supported if the knowledge engineer anticipates the exact generic instantiation and makes the instantiated subprogram objects available to the Access user. To better support generics, the system could be enhanced to support `GENERIC-SUBPROGRAM` objects which will create `SUBPROGRAM` objects when the user supplies the instantiation values.

In addition these conceptual enhancements to the existing ESL system, there are other enhancements that might be made to make the tool easier to use. The ESL Editor Panel and the ESL Editor Control Panel together constitute one logical group of functionality; the ESL Editor Control Panel displays information about the graph

edited on the ESL Editor Panel. The tool would be easier to use if these two panels were merged into one.

The system displays the name for a node below the image for the node on the ESL Editor Panel, but does not allow the user to change the name on this panel. To change the name for a node, the Node Details Panel must be displayed and the name be changed in the Name Field. Instead of using this mechanism to change node names, the tool would be easier to use if the node names could be changed directly on the ESL Editor Panel.

6.2 Developing Reusable Software Libraries

Another direction for future work on this system is in the design and development of reusable software libraries. As has been mentioned earlier, the usefulness of this tool is heavily dependent on the existence of a good function library. Work needs to be done both in describing how such libraries should be designed, and also in developing good libraries for various domains.

7. Conclusions

Extension of the ACCESS software to include the ESL editor has demonstrated the viability of a tool that can be used by non-programmers to create programs by direct manipulation of graphical components. The directed graph paradigm used to define programs is not specific to a particular problem domain and the system can be used by non-programming users in a great variety of fields. Presently, plans are for the system to be used to develop mission planning simulation software.

Although the tool can be used to express programs in any problem domain, the use of directed graphs to create programs is more appropriate for some types of problem domains than others. Nodes in these directed graphs can take input and produce output, and all data must be processed by being passed from node to node. The user has no means of explicitly defining and using local variables in a graph. These semantics make the tool better suited for problem domains in which data is passed through a succession of "processors" than one in which applications require a significant control structure. The ESL Editor supports IF nodes and ITERATOR nodes to allow branching and iteration, but these constructs tend to require the end user to have a fair amount of programming knowledge.

For the tool to be of most benefit, the application libraries made accessible by the system should be designed to be used with such a system. In particular, the inputs and outputs for the various software components should be of consistent data types. If they are not, the knowledge engineer will need to supply the user with an additional library of components that convert one data type to another or components that access objects contained in or pointed to by an object. Graphs constructed with such components will naturally be less readable because of the additional clutter introduced by the new components. The best type of library for use with this system is one that in which the components are designed to fit together, with the outputs produced by one group of component being directly mappable to the inputs accepted by other components. The set of functions provided by the library must also constitute a complete set of tools that can be used to address a wide variety of problems in an application domain.

Appendix A Graph Validation Algorithm

This appendix describes in some detail the algorithm which is used to check the validity of an ESL graph.

Some requirements for validity are enforced as the graph is created - e.g., that connectors connect ports of the same data type. However, most requirements will have to be checked at graph validation time.

The fundamental requirement which must be checked for graph validity is that data is produced at all graph output ports regardless of the path through the graph that is actually executed; in particular, regardless of the input data. In checking graph validity, it will be assumed that any given path through the graph has the potential to be executed - that is that any IF node can generate a THEN or an ELSE output.

As an example of the implications of this consider the following:

Suppose an IF node checks if a value is less than 0 and that when it is, the IF node triggers a IF node, which also accepts the same value through its input port as the IF node did. Then the graph validator will assume that both branches through the IF node are possible, even if one of them corresponds to a situation in which the input data is greater than 0. Thus, potentially, there are situations where the user knows that a particular path cannot occur, but the validator cannot identify that path.

During validation, one requirement which can be checked locally is that for each non-trigger input port on a node, there exists a connector with this port as destination. A second condition is that all control nodes (IF or LOOP) have connectors connected to each of their output ports.

Now, the steps in validating an ESL graph are as follows:

1. Check local conditions, i.e., that there is a connector to every node input port and a connector from every control node output port.

2. Identify nodes within the scope of each loop. The initial implementation will not support nested loops.

3. Label graph nodes in the order in which code will be generated and label nodes and connectors with the execution path (branches on control nodes) which is used to reach them. The algorithm for doing this is now described in some detail below

The motivation for this algorithm to label nodes in the order in which they will be

executed (also the order in which code will be generated) and to label each connector in such a way that one can identify the execution path which would have to be taken through the graph in order to produce data on that connector.

The node labels should be constructed in such a way that all nodes within the scope of a THEN branch of an IF construct are labeled sequentially, followed by all nodes within the scope of the ELSE branch.

The execution path labels consist of sequences which define the paths taken through the various control nodes. For example, if an execution path passes through IF node 12, producing a trigger output through the THEN branch, and then passes through IF node 17 producing a trigger output on the ELSE branch, then this path will be labeled ((12,1) (17,2)).

Once labels have been supplied for all connectors which connect the input ports of a given node, the algorithm then involves making sure that these labels are consistent. The first step consists in verifying consistency of labels on connectors to individual input ports of the node - both the data ports and the trigger input port. There is some difference in the way consistency is handle for these different input port types.

Ports on a particular node may be non-merge or merge ports. The first corresponds to the case in which there is a single connector with this port as destination; the second corresponds to the case in which there are multiple such connections.

If a node port is merge port for the node, then the algorithm consists of determining whether the labels for the connectors into that port can be merged consistently into a single label. There are two different types of merges which can occur at a merge port. The first type represents a true merge of data computed on disjoint execution paths from a control node. In this case, two execution paths labeled by sequences of the form ((N, 1)) and ((N, 2)) get merged to (). The meaning is that data is generated at this port for any execution path which goes through the THEN (1) or ELSE (2) branch of node N, so that data is generated at this port no matter which branch is taken through the IF node.

Another type of merge at a node data port is what is called an overwrite merge. Suppose, for example, if one connector to the port is labeled with the empty sequence and another is labeled with the sequence ((N, i). This could represent the case in which a constant value is supplied as a default input to that port and a different value is followed if the execution path that goes on branch i from node N is followed. Thus data is available at the port no matter which execution path is followed. Thus the label which would be attached to the merged data is the empty sequence. In general, if there are two execution path labels, one a subsequence of the other, an overwrite merge merges them to the shorter sequence.

For merges at node data ports, the algorithm will perform true execution path merges first and then perform overwrite merges. If an execution path merge occurs at a node, this must be recorded, as it affects the determination of consistency among the input ports in a way which will be described below.

NOTE: A consequence of the above definitions is that data generated by nodes within the different branches of IF construct which is in itself within the scope of another control construct be merged if necessary prior to being merged with data from a different branch of the parent control construct. For example, the sequences ((N, i)) and ((M, j)) will not be merged. In short, the design decision has been made not to carry along a label which represents the union of these two execution paths. This limitation corresponds to good programming practice.

Each node has a trigger input port. The purpose of this input port is to allow sequencing of node execution. Since the algorithm being described ensures that a node will not be labeled until the connectors to its trigger input port have been labeled, there will be no consistency check for labels on connectors to this node.

If all connectors into a node are labeled by execution path and merging of these labels at merge ports has occurred successfully, then one must validate consistency between the labels which have been computed for the input ports. The following conditions must be satisfied:

1. All new labels which have been computed for data input ports for this node must be an initial segment of the longest sequence labeling any of the connectors. For example, the labels ((12, 1) (17, 2) (20 1)), ((12,1)), and ((12, 1) (17, 2})) for three connectors connecting to the three input ports of a node are consistent - data will be available at this node when the execution path corresponding to ((12, 1) (17, 2) (20 1)) is followed. However, the labels ((12, 1)), ((12, 1) (17, 2)), and ((12, 2) (17, 2) (20 1)) are not consistent, because the third label shows data reaching this port via a different branch out of control node 12 than do the first two labels.

2. However, if one or more execution path merge has occurred at any of the input ports, the labels corresponding to these merges must all represent the same and most specific execution path labels attached to any of the node input ports.

   For example, this means that if data at one node is available only along the execution path ((N, 1), then there should not be a merge at another node of data from ((N, 1)) and ((N,2)). However, data computed before the IF node branch (i.e., supplied along a connector with () as label), can be merged with data with the label ((N, 1).

The following is an example of graph which would not be allowed under this interpretation:

If the labels computed for the various input ports of a node are consistent as described above, the labels merge to the label representing the most specific execution path - e.g., to the label represented to the longest sequence.

The following in more detail is the algorithm for labeling nodes and connectors:

Make the pseudo node corresponding to the graph input ports eligible to be labeled. Set the execution path label for this node to the empty sequence. Set the next node label to 1.

Do until done:

- Take highest priority node eligible to be labeled and label with the next node label, M, then increment the value of the next available node label. Label the connectors from this node as follows:

  1. If this is an IF node, the output ports from this node are numbered (1 for THEN, 2 for ELSE. The label on the connectors consists of the connector label for the node with (N, i) appended to the end of the sequence, where i is the number of the output port to which the connector connects. For example, if an IF node were labeled ((K, 1)), the connector connecting to its THEN output port would be labeled ((K,1), (M, 1)) and the connector connecting to its ELSE output port would be labeled ((K, 1) (M, 2)).

  2. If this is not a control node, then label the connectors with the execution path label for this node.

- After the connectors from the node have been labeled, redetermine the nodes eligible to be labeled. Do this by iterating over all nodes to which the newly labeled node is connected.

  - The graph is not valid if any such node is already labeled, unless the two nodes are within the scope of the same loop.

  - For each node to which the newly labeled node is connected, determine whether all connectors connecting to the ports corresponding to this struct have been labeled. The exception is that if a port on a node within the scope of a loop is a merge port, only those connectors from outside the loop must be labeled. Also, only the connector to the INITIAL port of an ITERATE node must be labeled in order to process the node. It is invalid for a label to be applied to a connector

to a LOOP port prior to labeling the ITERATE node. If all required connectors to the node have been labeled, then determine consistency of the labels as described above. If consistency does not hold at any input port or for the node as a whole, then the graph is invalid. If consistency holds, compute the execution path label for the node.

- If a merge occurs at a node port within the scope of a loop, the label created must be a subsequence of the label on the INITIAL input port of the ITERATOR node at the top of the loop. This ensures that data supplied within the scope of the loop does not come from an execution path more specific than the one which triggered the start of the loop.

- If the newly computed label is for an input port for a node outside the scope of a loop, it must not depend on any node within the scope of the loop.

- If a node has been assigned an execution path label, then it is eligible to receive a node label. The exception is that the ITERATOR node for a loop is not eligible to receive a node label until all input ports within the loop which receive input from outside the loop have received labels.

- Now, label the highest priority node eligible to be labeled. Priority is determined on the basis of the execution path label for the node and the type of the node, as follows:

  - Once the ITERATOR node for a loop has received a label, any node within the same loop has priority over a node outside the loop.

  - A node corresponding to a label containing the pair (N, 1) is assigned a node label before one containing the pair (N, 2) (i.e., nodes along the THEN branch of an IF node are assigned values before ones along the ELSE branch).

  - Once a node whose execution path label contains the pair ((N, i)) has been labeled, any node whose execution path label contains this pair has priority over a node whose execution path does not contain this pair.

  - If two nodes correspond to the same label, a procedure or subgraph node takes precedence over a control node.

  - Finally, precedence is determined by the physical representation of the graph with leftmost and topmost nodes having priority.

End Do

The process is complete when there are no nodes left to be labeled.

The following conditions must be satisfied in order for the graph to be valid:

1. All nodes must have been labeled.

2. The connectors to the graph output ports must be labeled with the empty sequence. This means output will be produced at all graph output ports regardless of the execution path which is followed.

3. The node labels on the nodes with an execution path label containing the pair (M, i) must represent a contiguous set of integers.

References

[Inference 91] Inference Corporation.
ART-IM 2.5 Reference Manual.
Inference Corporation, 1991.

[InterViews 91] Mark Linton.
InterViews Reference Manual, Version 3.1.
Stanford University, 1991.

[Softech 90] Softech, Inc.
Summary Report for the Engineering Scripting Language (ESL).
Technical Report Subcontract SE.33, NCC-9-16, Report to NASA and University of Houston-Clear Lake, 1990.

[TAE Plus 91] NASA.
TAE Plus User Interface Developer's Guide, Version 5.1.
NASA, 1991.